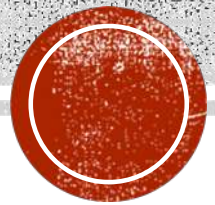

Informatics Engineering, University of Mataram.

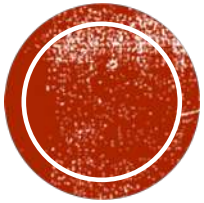
# INHERITANCE
# (Multiple Inheritance)

Abstract Class

Interface

Inner Class





# KELAS ABSTRAK

Abstract method, Abstract Class

# ABSTRACT METHOD

- Method that has no implementation (body)
- To create an abstract method, only write the method declaration without the body, and use the abstract keyword
  - Without → { }
- Example :

```
//without{ }
public abstract void someMethod();
```

# ABSTRACT CLASS

- Abstract class is a class that has at least one abstract method
- Used when the parent class that is the source of inheritance does not want to be processed.
- Abstract class cannot be instantiated (cannot be objectized)

```
//The code below causes a compilation error
MyAbstractClass a1 = new MyAbstractClass();
```

- To use an abstract class, another class is used (concrete class)
- Concrete class must implement all abstract methods
- Concrete class uses extends keyword

# EXAMPLE : ABSTRACT CLASS

```
public abstract class MakhlukHidup {
    public void bernafas(){
        System.out.println("Makhluk hidup bernafas...");
    }
    public void makan(){
        System.out.println("Makhluk hidup makan...");
    }
    /**
    * Abstract method berjalan()
    * Kita ingin mengimplementasi method ini melalui Concrete Class
    */
    public abstract void berjalan();
}
```

# EXTENDING ABSTRACT CLASS

- When a concrete class implements the abstract class MakhlukHidup, the concrete class must implement (override) the berjalan() method. Otherwise, the class will become an abstract class too, so it cannot be instantiated/created object.
- Example

```
public class Manusia extends MakhlukHidup {
    public void berjalan(){
        System.out.println("Manusia berjalan...");
    }
}
```

# WHEN USE ABSTRACT METHOD AND ABSTRACT CLASS?

- Abstract methods are used when two or more subclasses are designed to fulfill the same rule with different implementations
  - These subclasses extend the same abstract class but with different implementations of the abstract methods.
- Using abstract classes as a general concept at the top level of the class hierarchy, and using many subclasses to create detailed implementations of abstract classes.

# EXAMPLE

```
public abstract class Printer {
    public abstract void printing();
    public void scanner(){
    }
}
```

```
public class Canon extends Printer{
    public void printing(){
        System.out.println("nge-print pakai tinta");
    }
}
```

```
public class Samsung extends Printer {
    public void fotocopy(){
        System.out.println("Bisa buat fotocopy");
    }
    public void printing(){
        System.out.println("nge-print pakai carbon");
    }
}
```

# EXAMPLE : ABSTRACT CLASS

**Hewan.java**

```
abstract class Hewan {
    protected String nama;
    protected int jumKaki;
    protected boolean bisaTerbang = false;
    public Hewan(String nama, int kaki, boolean terbang) {
        this.nama = nama;
        jumKaki = kaki;
        bisaTerbang = terbang;
    }
    public abstract void bersuara();
    public  static void  makan() {
        System.out.println("nyam, nyam, nyam");
    }
    public void isHewan() {
        System.out.println("nama : "+nama);
        System.out.println("jumlah kaki : "+jumKaki);
        System.out.println("bisa terbang : "+bisaTerbang);
    }
}
```

```
class Perkutut extends Hewan {
    public Perkutut() {
        super("perkutut",2,true);
    }
    public void bersuara() {
        System.out.println("\ncuit, cuit, cuit");
    }
    public static void main(String[] args) {
        Perkutut p = new Perkutut();
        p.isHewan();
        p.bersuara();
    }
}
```

**Output ?**

## Perkutut.java

```
class Perkutut extends Hewan {
    public Perkutut() {
        super("perkutut",2,true);
    }
    public void bersuara() {
        System.out.println("\ncuit, cuit, cuit");
    }
    public static void main(String[] args) {
        Perkutut p = new Perkutut();
        p.isHewan();
        p.bersuara();
    }
}
```

**Output :**

```
nama : perkutut
jumlah kaki : 2
bisa terbang : true

cuit, cuit, cuit
```

## Sapi.java

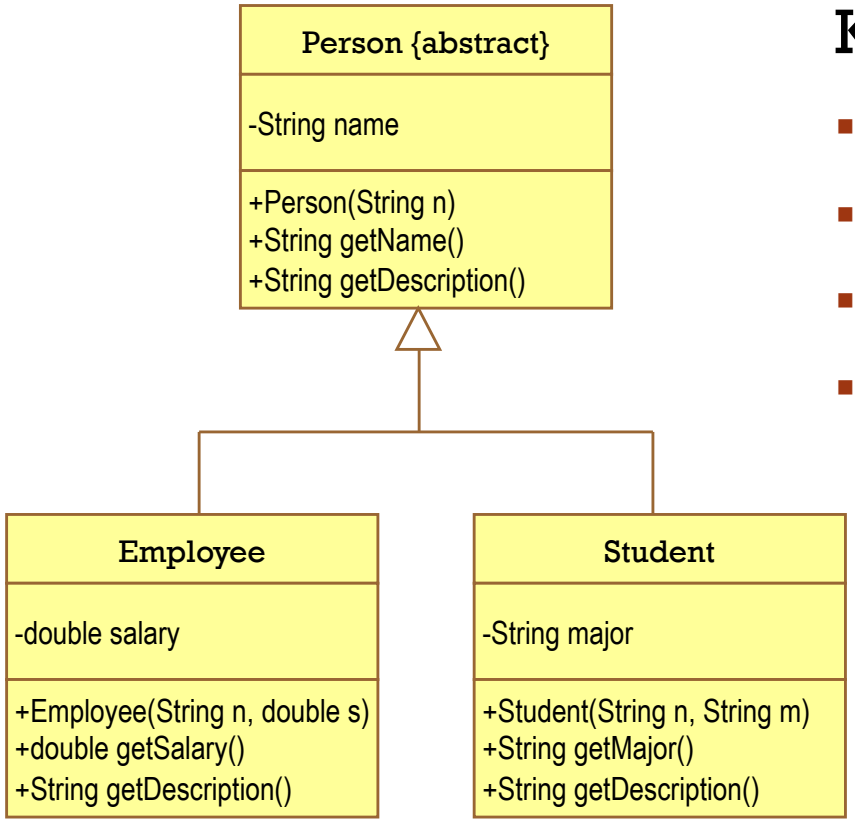
```
class Sapi extends Hewan {
    public Sapi() {
        super("sapi", 4, false);
    }
    public void bersuara() {
        System.out.println("\nemoh…,emoh..");
    }
    public static void main(String[] args) {
        Sapi s = new Sapi();
        s.isHewan();
        s.bersuara();
    }
}
```

**Output :**

```
jumlah kaki : 4
bisa terbang : false

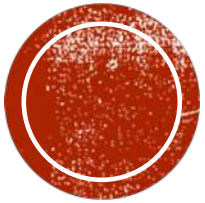
emohà,emoh..
```

# EXAMPLE OF ABSTRACT CLASSES :




## Kode Program

- Person.java
- Employee.java
- Student.java
- CobaKelasAbstrak.java



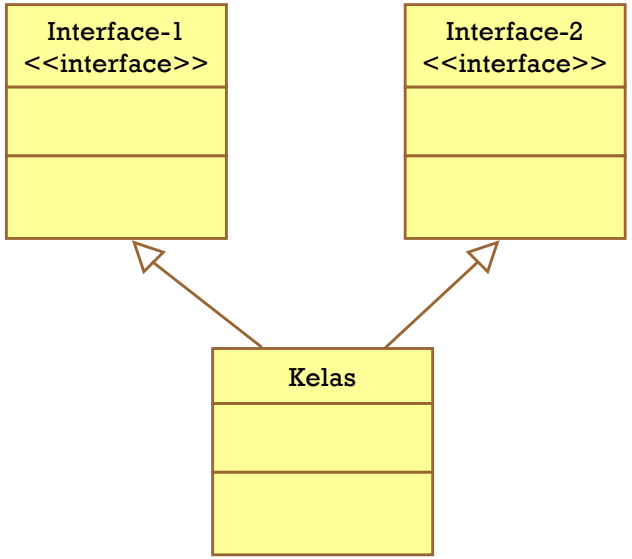
# MULTIPLE INHERITANCE AND INTERFACE

# MULTIPLE INHERITANCE IN JAVA

- Java ***does NOT*** support multiple inheritance with 'extends'
- Java ***ONLY*** has *single inheritance* with 'extends'
  - A derived class has only ***one parent class***
- Then How is it realized in Java? Java realizes it by using *Interface*

# MULTIPLE INHERITANCE IN JAVA : INTERFACE

- A class can inherit from multiple interfaces (multiple inheritance).




```
class Kelas implements
Interface1, Interface2 {

  // Atribute

  // Method

}
```

# INTERFACE

- An interface is used when we want to specify what a class should do but not how to do it.

- *Interface ( < java 8)* is a collection of function declarations (without implementation). Interface can also declare constants

- Interface *( < java 8)* is actually the same as a class, but only has method declarations without implementation

# INTERFACE DECLARATION

Interface Declaration:

```
[modifier] interface NamaInterface {
    // deklarasi konstanta
    // deklarasi method
}
```

# INTERFACE IMPLEMENTATION

```
[modifier] class NamaKelas implements NamaInterface {
    // penggunaan konstanta
    // implementasi method
}
```

# INTERFACE ( < JAVA 8)

- A class prototype that contains **constant definitions** and **method declarations** (only method names without program code definitions).

```
interface Perusahaan {
  // Konstanta
  String KOTA = "Bandung";

  // Deklarasi method
  void setNama(String n);
  void setAlamat(String a);
  String getNama();
  String getAlamat();
}
```

# INTERFACE ( < JAVA 8)

- In an interface:
  - All constants (attributes) are **public**, **static** and **final**.
  - All methods are **abstract** and **public**.
  - No constructor declarations.

- Interfaces are used to express the **functional specifications** **of** several classes in general.

# INTERFACE

- Method definition is done in the class that implements the interface.

```
class Telkom implements Perusahaan {
  // Deklarasi atribut
  private String nama, alamat;

  // Definisi method
  public void setNama(String n) {
    nama = n;
  }
  ...
}
```

# EXAMPLE OF INTERFACE

- Interface : StartStop.java

```
interface StartStop {
  void start();
  void stop();
}
```

# EXAMPLE OF INTERFACE

- Implementation Class : UseStartStop.java

```
class Car implements StartStop {
  private String name;

  Car (String name) {
    this.name = name;
  }

  public void start() {
    System.out.println ("Insert key into ignition and turn.");
  }

  public void stop() {
    System.out.println ("Turn key and remove from ignition.");
  }

  public String getName() {
    return name;
  }
}
```

# EXAMPLE OF INTERFACE

- Implementation Class : UseStartStop.java

```
class WashingMachine implements StartStop {
  private String name;

  WashingMachine (String name) {
    this.name = name;
  }

  public void start() {
    System.out.println ("Push button and turn right.");
  }

  public void stop() {
    System.out.println ("Turn left and push button.");
  }

  public String getName() {
    return name;
  }
}
```

# INTERFACE

- Implementation Class : UseStartStop.java

```
class UseStartStop {
  public static void main (String [] args) {
    Car c = new Car("Impala");
    System.out.println(c.getName());
    System.out.print("Start: ");
    c.start();
    System.out.print("Stop : ");
    c.stop();

    WashingMachine w = new WashingMachine("Tzu Zian");
    System.out.println("\n"+w.getName());
    System.out.print("Start: ");
    w.start();
    System.out.print("Stop : ");
    w.stop();
  }
}
```

# INTERFACE VS ABSTRACT CLASS

| | *Variables* | *Constructors* | *Methods* |
|---|---|---|---|
| Abstract Class | Free, no restrictions, no restiction | Constructors can be called via subclasses through the constructor chain.<br>No object can be created. | Free, no restrictions,<br>At least one abstract method |
| Interfaces | All variables must be declared<br>public static final | There is no constructor.<br>No objects can be created. | All functions must be declared<br>public abstract |

# INTERFACE VS ABSTRACT CLASS

- Java only allows *single inheritance* for classes, but *multiple* for *interfaces*.

```
public class NewClass extends BaseClass implements Interface1, ..., InterfacesN {
  // ....
}
```

- An interface can be derived from another interfaces class by using the extends *keyword*.

```
public interface NewClass extends Interface1, ..., InterfacesN {
  // konstanta dan abstract method
}
```

# INHERITANCE ON INTERFACE

- Interfaces can have inheritance relationships between themselves. Using the **"extends"** keyword.
- Example :

  interface PersonInterface is superinterface from student interface.

```
public interface PersonInterface {
      . . .
}

public interface StudentInterface extends PersonInterface
{
      . . .
}
```

# IMPLEMENTS SUBINTERFACE

- A class that implements a subinterface must redeclare all methods in the subinterface as well as those in the superinterface.

- Example:

```
Class Mahasiswa implements StudentInterface {
      . . .
}
```

- Because **StudentInterface** is a subclass of PersonInterface, the **Student** class must redeclare all abstract methods owned by both interfaces.

# MULTIPLE-IMPLEMENTS

```
public interface PersonInterface {
    public void setNama(String nama);
}
```

```
public interface StudentInterface extends
PersonInterface{
    public void setSekolah(String sekolah);
}
```

```
public abstract class Mahasiswa implements StudentInterface
{

    public abstract void isiBiodata(String nama, int nilai);

    public void setSekolah(String skul){

    }
    public void setNama (String nam){

    }
}
```

# INTERFACE EVOLUTION IN JAVA

## Java 7

All methods are implicitly abstract (no body). No method implementations allowed. Only constants (public static final).

```
interface Animal {
   void eat();   // abstract
   void sleep(); // abstract
}
```

## Java 8

- Default methods → provide standard implementation, can be overridden.
- Static methods → called via interface name.

```
interface Animal {
    void eat(); // abstract
    default void sleep() {
        System.out.println("Sleeping...");
    }
    static void info() {
        System.out.println("All animals share traits.");
    }
}
```

## Java 9

Additional features:

- private methods → used internally in the interface.
- Useful for sharing code among default/static methods.

```
interface Animal {
    private void log(String msg) {
        System.out.println("Log: " + msg);
    }
    default void sleep() {
        log("Sleeping...");
    }
}
```

# JAVA 7 EXAMPLE (ABSTRACT METHODS ONLY)

- All methods in interface are abstract.
- Subclass Cat must implement all methods.
- ```
  interface Animal {
      void eat();
      void sleep();
  }

  class Cat implements Animal {
      @Override
      public void eat() {
          System.out.println("Cat eats fish.");
      }

      @Override
      public void sleep() {
          System.out.println("Cat sleeps on the sofa.");
      }
  }
  ```


  ```

# JAVA 8 EXAMPLE (DEFAULT & STATIC METHODS)

- Interface can have default and static methods.
- Subclass Cat may override default method.
- 

```
interface Animal {
    void eat();

    default void sleep() {
        System.out.println("Sleeping in default way...");
    }

    static void info() {
        System.out.println("All animals need food and rest.");
    }
}

class Cat implements Animal {
    @Override
    public void eat() {
        System.out.println("Cat eats fish.");
    }

    @Override
    public void sleep() {
        System.out.println("Cat sleeps on the sofa.");
    }
}
```

**default keyword in interfaces (since Java 8)**

•This is **not an access modifier**.
•In an interface, default means that a method has a **built-in implementation** (a body).
•A class that implements the interface can **use it as is** or **override it**.

```java
interface Animal {
    void eat(); // abstract method

    // default method with built-in implementation
    default void sleep() {
        System.out.println("Sleeping in default way...");
    }
}

class Cat implements Animal {
    @Override
    public void eat() {
        System.out.println("Cat eats fish.");
    }
    // Cat does not override sleep(), so it will use the default implementatio
}

public class Main {
    public static void main(String[] args) {
        Animal a = new Cat();  // create object
        a.eat();               // calls overridden method in Cat
        a.sleep();             // calls default method from interface
    }
}
```

# JAVA 8 EXAMPLE (DEFAULT & STATIC METHODS)

**default keyword in interfaces (since Java 8)**

- This is **not an access modifier**.
- In an interface, default means that a method has a **built-in implementation** (a body).
- A class that implements the interface can **use it as is** or **override it**.

# JAVA 9 EXAMPLE (PRIVATE METHODS IN INTERFACE)

- Private methods help share code inside interface.
- Subclass Cat cannot access private methods directly.
- ```
  interface Animal {
      void eat();

      private void log(String msg) {
          System.out.println("Log: " + msg);
      }

      default void sleep() {
          log("Animal is sleeping...");
      }
  }

  class Cat implements Animal {
      @Override
      public void eat() {
          System.out.println("Cat eats fish.");
      }
  }
  ```


  ```

```java
interface Animal {
    void eat(); // abstract method

    // private helper method (cannot be accessed by subclasses directly)
    private void log(String msg) {
        System.out.println("Log: " + msg);
    }

    // default method that reuses private method
    default void sleep() {
        log("Animal is sleeping...");
    }
}

class Cat implements Animal {
    @Override
    public void eat() {
        System.out.println("Cat eats fish.");
    }

    // ❌ This would cause an error:
    // public void test() {
    //     log("Trying to call private method"); // not allowed
    // }
}

public class Main {
    public static void main(String[] args) {
        Animal a = new Cat();
        a.eat();    // from Cat
        a.sleep();  // default method calls private helper inside interface
    }
}
```

# JAVA 9 EXAMPLE (PRIVATE METHODS IN INTERFACE)

# DISCUSSION

Diskusi berdua → Class diagram sederhana + kode Tunjukkan:

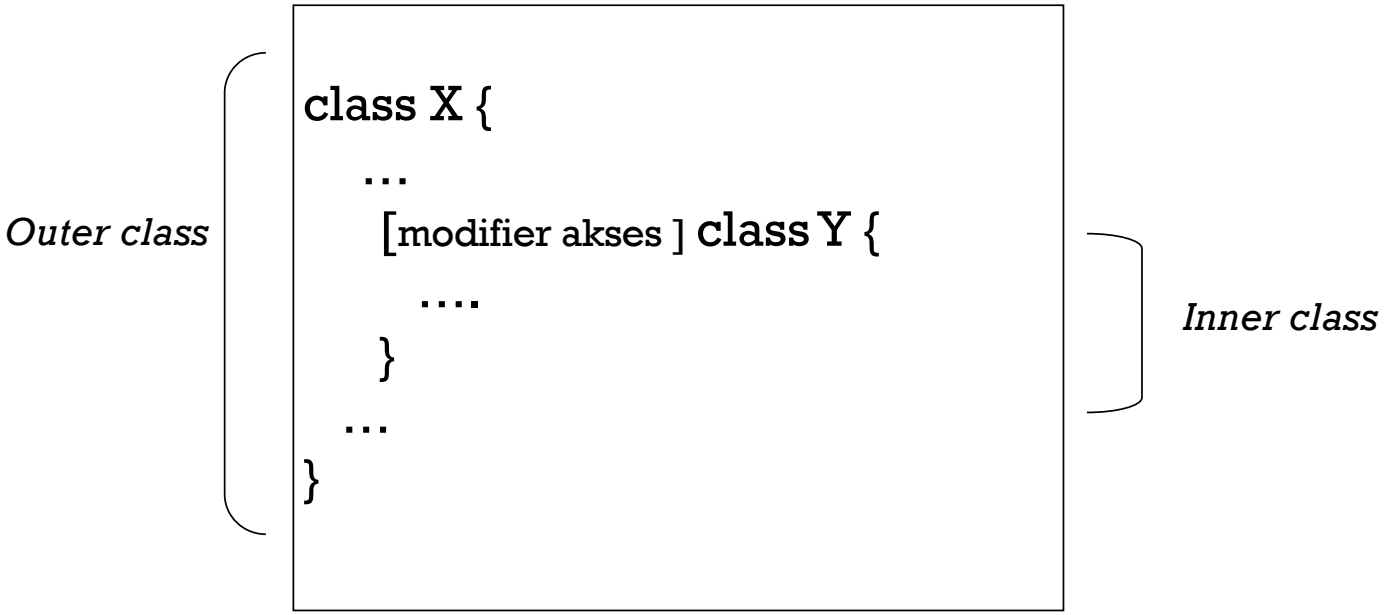
1. Inheritance
2. Polymorphism
3. Interface/Abstract class.


- Application Programming Interface

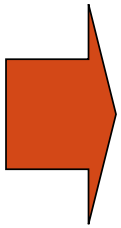
# INNER CLASS

# INNER KELAS

- Classes that are declared inside a class




- Generates two files *.class, that is *Outer.class* dan *Outer$Inner.class*



X.class
X$Y.class

# INNER KELAS

- Inner class is part of a class, so it is treated like any other class member
- Inner can access all members of the outer class, and vice versa
- Inner class is used like a normal class, but it can be used outside its outer class depending on the access modifier.

# EXAMPLE OF INNER KELAS

```
class Buku {
     private int nomor;
     private String judul;
     class Bab {
        private int noBab;
        public String getBab() {
          return "nomor buku: "+nomor+", judul buku: "+judul+", nomor Bab: "+noBab;
        }
     }
     private Bab bab = new Bab();
     public void setBab(int nomor,String judul,int noBab ) {
        this.nomor = nomor;
        this.judul = judul;
        bab.noBab = noBab;
     }
     public void cetak(){
        System.out.println(bab.getBab());
     }
}
```

# EXAMPLE OF INNER KELAS

```
class DemoBuku {
    public static void main(String[] args) {
      Buku buku = new Buku();

      buku.setBab(10,”PBO 1”,6)
      buku.cetak();
    }
}
```

# THANK YOU